CS:5810 Formal Methods in Software Engineering

Alloy Modules

Alloy Modules

- Alloys has a module system that allows the modularization and reuse of models
- A module defines a model that can be incorporated as a submodel into another one
- To facilitate reuse, modules may be parametric in one or more signatures

Examples

```
module util/relation
 -- r is acyclic over the set s
 pred acyclic [r: univ->univ, s: set univ] {
   all x: s | x !in x.^r
 }
```

```
module family
 open util/relation as rel
 sig Person {
   parents: set Person
 }
 fact { acyclic[parents, Person] }
```

Examples

```
module util/relation
 -- r is acyclic over the set s
 pred acyclic [r: univ->univ, s: set univ] {
   all x: s | x !in x.^r
 }
```

```
module fileSystem
 open util/relation as rel
 sig Object {}
 sig Folder extends Object {
   subFolders: set Folder
 }
 fact { acyclic[subFolders, Folder] }
```

Module Declarations

- The first line of every module is a module header

```
module modulePathName
```

- The module can import another module with an **open** statement immediately following the header

```
open modulePathName
```

Module Definition

- A module A can import a module B which can in turn import a module C, and so on
- You can understand open statements informally as textual inclusion
- No cycles in the import structure are permitted

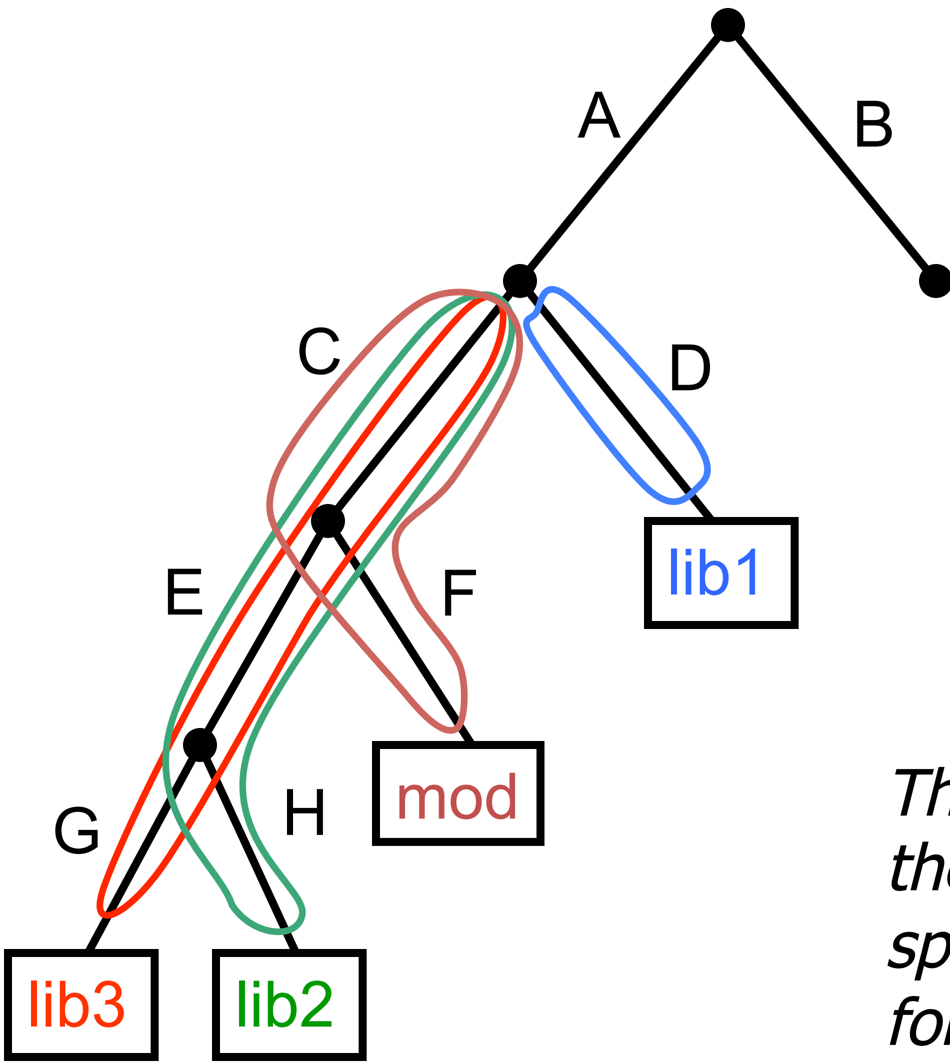
ModulePathName Definition

- Every module has a path name that must match the path of its corresponding file in the file system

- The module's path name can range
  - from just the name of the file (without the .als extension)
  - to the whole path from the root

- The root of the path in the importing module header is the root of the path of every import

Examples



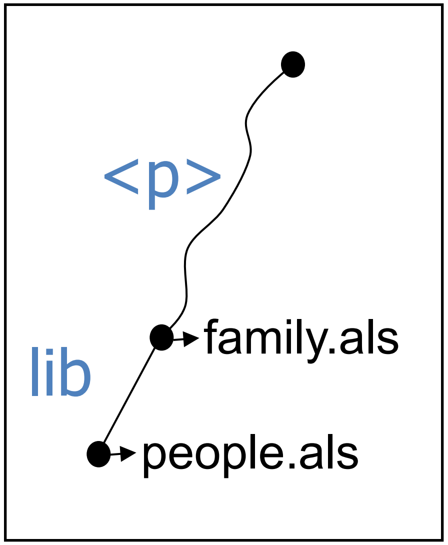
model C/F/mod
open D/lib1
open C/E/H/lib2
open C/E/G/lib3

The modulePathName in the module header just specifies the root directory for every imported file

ModulePathName definition

- Example:

```
module family
  open lib/people
```

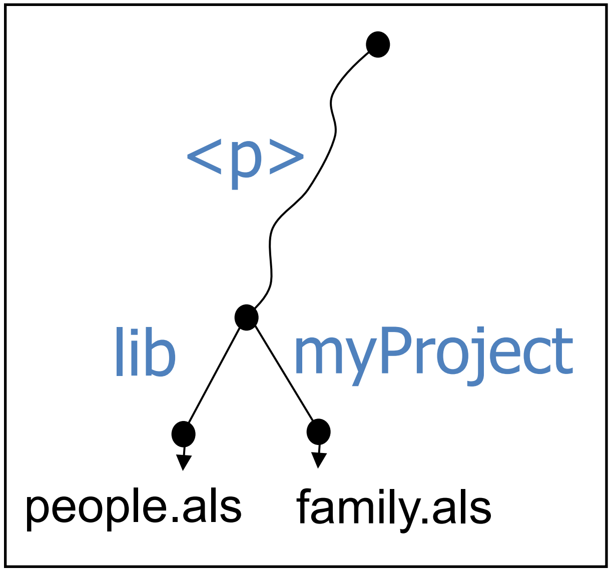


- If the path of `family.als` is `<p>` in the file system then the Alloy Analyzer will search `people.als` in `<p>/lib/`

ModulePathName definition

- Example:

```
module myProject/family
  open lib/people
```



- If the path of `myProject` is `<p>` in the file system then AA will search `people.als` in `<p>/lib/`

Predefined Modules

- Alloy 4 comes with a library of predefined modules
- Any imported module will actually be searched first among those modules
  - Examples:
    - `book/chapter2/addressBook1a`
    - `util/relation`
    - `examples/puzzles/farmer`
- Failing that, the rules in the previous slides apply

As

- When the path name of an import includes / (*i.e.* it is not just the name of a file but also a path)
- Then you may give a shorter name to the module with **as**

```
open util/relation as rel
```

Name Clashes

- Modules have their own namespaces
- To avoid name clashes between components of different modules, we use qualified names

```
module family
 open util/relation as rel
 sig Person { parents: set Person }
 fact { rel/acyclic [parents] }
```

Parametric Modules

- A model `m` can be parametrized by one or more signature parameters `[x1,…,xn]`

- Any importing module must instantiate each parameter with a signature name

- The effect of opening `m[S1,..,Sn]` is that of importing a copy of `m` with each signature parameter `xi` replaced by the signature name `Si`

Parametric Modules Example

```
module graph[node] // 1 signature param
 open util/relation as rel

 pred dag[r: node -> node] {
   rel/acyclic[r, node]
 }


module family
 open util/graph[Person] as g
 sig Person { parents: set Person }
 fact { dag[parents] }
```

The Predefined Module Ordering

- Creates a single linear ordering over the atoms in `S`

```
module util/ordering[S]
```

- It also constrains all the atoms to exist that are permitted by the scope on `S`
  - If the scope on a signature `S` is 5, opening `ordering[S]` will force `S` to have 5 elements and create a linear ordering over those five elements

The Module Ordering

```
module util/ordering[S]
private one sig Ord {
    First, Last: S,
    Next, Prev: S -> lone S
}
fact {
   // all elements of S are totally ordered
   S in Ord.First.*Next
   ...
}
```

The Module Ordering

```
// constraints that actually define the
// total order
Ord.Prev = ~(Ord.Next)
one Ord.First  // redundant with signature decl.
one Ord.Last   // redundant with signature decl.
no Ord.First.Prev
no Ord.Last.Next
```

The Module Ordering

```
//
//
(one S and no S.(Ord.Prev) and no S.(Ord.Next))
or
//
all e: S |
  //
  //
  (e = Ord.First or one e.(Ord.Prev)) and

  //
  //
  (e = Ord.Last or one e.(Ord.Next)) and

  //
  (e !in e.^(Ord.Next))
```

The Module Ordering

```
// either S has exactly one atom,
// which has no predecessors or successors ...
(one S and no S.(Ord.Prev) and no S.(Ord.Next))
or
// or ...
all e: S |
  // ... every element except the first has one
  // predecessor, and ...
  (e = Ord.First or one e.(Ord.Prev)) and

  // ... every element except the last has one
  // successor, and ...
  (e = Ord.Last or one e.(Ord.Next)) and

  // ... there are no cycles
  (e !in e.^(Ord.Next))
```

The Module Ordering

```
//
fun first: one S { Ord.First }
//
fun last: one S { Ord.Last }
//
//
fun prev [e: S]: lone S { e.(Ord.Prev) }
//
//
fun next [e: S]: lone S { e.(Ord.Next) }
//
fun prevs [e: S]: set S { e.^(Ord.Prev) }
//
fun nexts [e: S]: set S { e.^(Ord.Next) }
```

The Module Ordering

```
// first
fun first: one S { Ord.First }

// last
fun last: one S { Ord.Last }

// return the predecessor of e, or empty set if e is
// the first element
fun prev [e: S]: lone S { e.(Ord.Prev) }

// return the successor of e, or empty set of e is
// the last element
fun next [e: S]: lone S { e.(Ord.Next) }

// return elements prior to e in the ordering
fun prevs [e: S]: set S { e.^(Ord.Prev) }

// return elements following e in the ordering
fun nexts [e: S]: set S { e.^(Ord.Next) }
```

The Module Ordering

```
// e1 is before e2 in the ordering
pred lt [e1, e2: S] { e1 in prevs[e2] }

// e1 is after than e2 in the ordering
pred gt [e1, e2: S] { e1 in nexts[e2] }

// e1 is before or equal to e2 in the ordering
pred lte [e1, e2: S] { e1=e2 || lt [e1,e2] }

// e1 is after or equal to e2 in the ordering
pred gte [e1, e2: S] { e1=e2 || gt [e1,e2] }
```

The Module Ordering

```
// returns the larger of the two elements in the
// ordering
fun larger [e1, e2: S]: S
         { lt[e1,e2] => e2 else e1 }

// returns the smaller of the two elements in the
// ordering
fun smaller [e1, e2: S]: S
         { lt[e1,e2] => e1 else e2 }

// returns the largest element in es
// or the empty set if es is empty
fun max [es: set S]: lone S
         { es - es.^(Ord.Prev) }

// returns the smallest element in es
// or the empty set if es is empty
fun min [es: set S]: lone S
         { es - es.^(Ord.Next) }
```